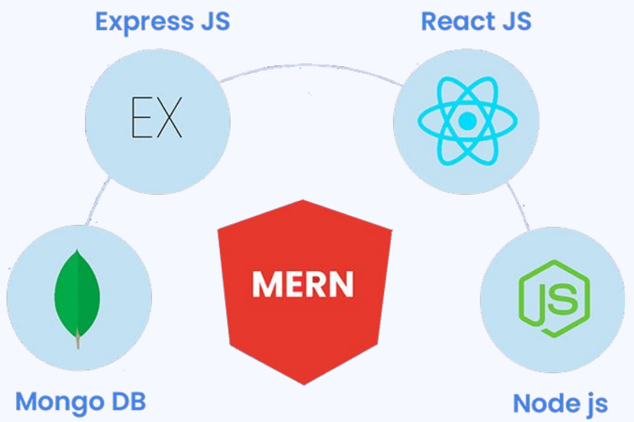


Full Stack Development Using MERN

Day 07 - 16 June 2024

Lecturer Details:

Chanaka Wickramasinghe
cmw@ucsc.cmb.ac.lk
0771570227

Prameeth Madhuwantha
bap@ucsc.cmb.ac.lk
0772888098

Anushka Vithanage
dad@ucsc.cmb.ac.lk
071 527 9016

Lecture 07

CRUD Operations, User Authentication, Authorization

Connecting MongoDB with the Backend

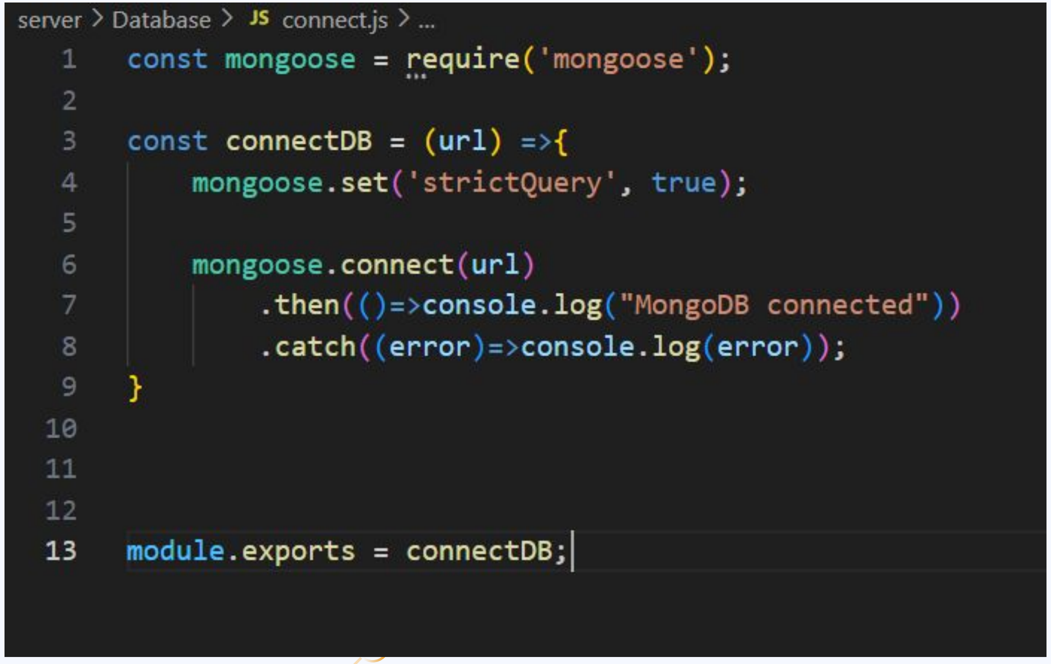
- npm install "mongoose"
  - Mongoose is a popular Object Data Modeling (ODM) library for MongoDB in Node.js, and it offers several advantages over using plain boilerplate MongoDB queries.

Advantages of Mongoose

- Schema Definition
- Simplified Queries
- Validation
- Middleware
- Built-in Promises
- Data Type Casting
- Plugins and Hooks

Connecting MongoDB with the Backend

- Create a new folder "Database"
- Create a new file as "Connect.js"


server > Database > JS connect.js > ...

```javascript
const mongoose = require('mongoose');

const connectDB = (url) =>{
    mongoose.set('strictQuery', true);

    mongoose.connect(url)
        .then(()=>console.log("MongoDB connected"))
        .catch((error)=>console.log(error));
}



module.exports = connectDB;
```

MongoDB Connection Ctd.

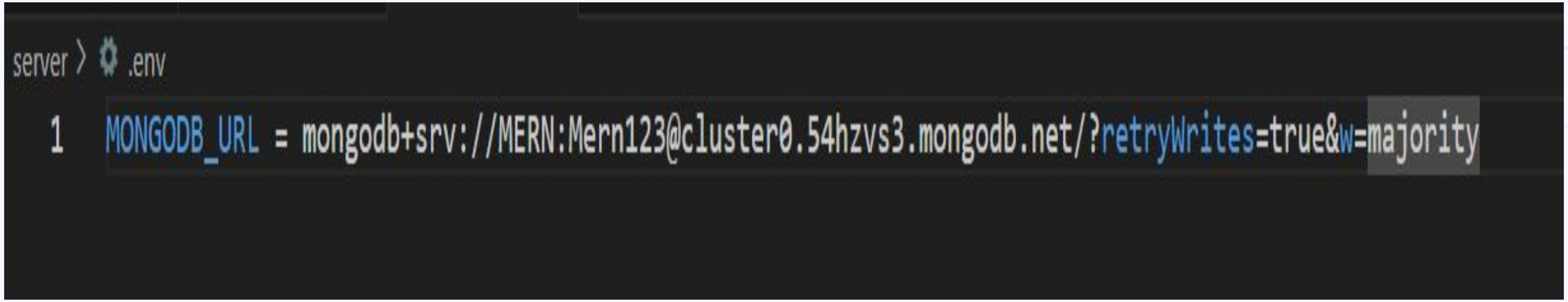
1. `const mongoose = require('mongoose');`
   - This line imports the `mongoose` module, which is an Object Data Modeling (ODM) library for MongoDB and provides a convenient way to interact with MongoDB databases using JavaScript.
2. `const connectDB = (url) => {`
   - This line declares a function named `connectDB` that takes a parameter `url`. This function is responsible for establishing a connection to a MongoDB database using the provided URL.
3. `mongoose.set('strictQuery', true);`
   - This line sets the Mongoose option `strictQuery` to `true`. When `strictQuery` is enabled, Mongoose will throw an error if a query contains undefined fields.
4. `mongoose.connect(url)`
   - This line initiates the connection to the MongoDB database using the provided `url`. It establishes a connection to the MongoDB server and returns a promise.

MongoDB Connection Ctd.

5. `.then(() => console.log("MongoDB connected"))`
   - This line registers a callback function that is executed when the `mongoose.connect()` promise is resolved successfully. In this case, it logs the message "MongoDB connected" to the console, indicating that the connection to the MongoDB database was successful.
6. `.catch((error) => console.log(error));`
   - This line registers a callback function that is executed when the `mongoose.connect()` promise is rejected or encounters an error. It logs the error message to the console, providing information about the error that occurred during the connection process.
7. `module.exports = connectDB;`
   - This line exports the `connectDB` function, making it available to other modules when they require or import this module. It allows other parts of the application to use this function to establish a connection to the MongoDB database.

MongoDB Connection Ctd.

- Create .env file

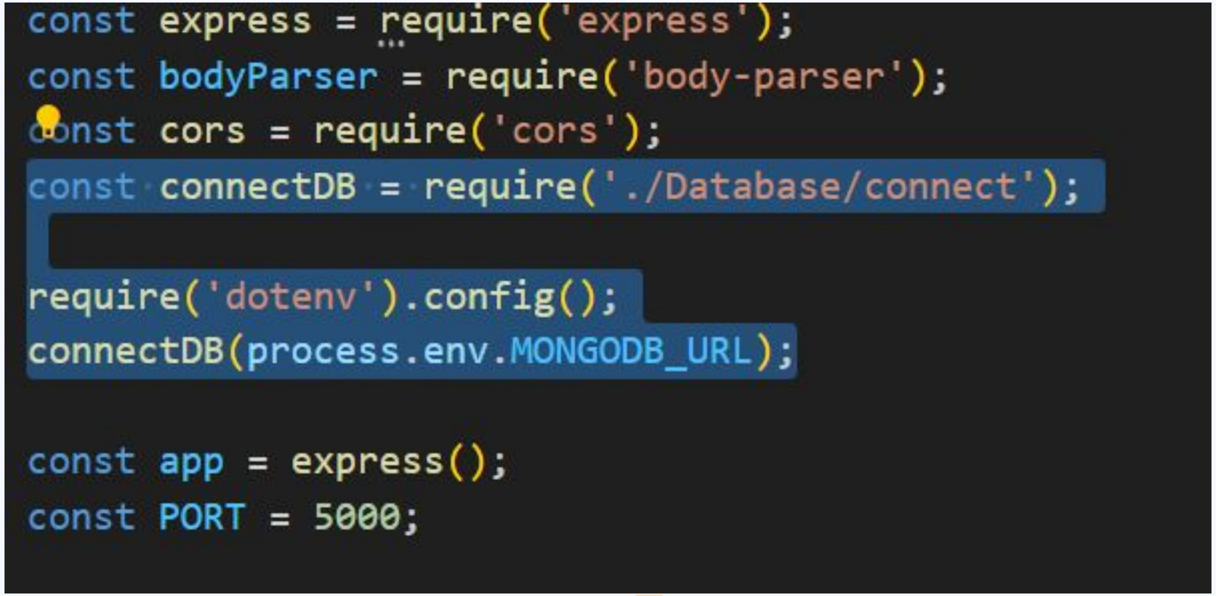


server > .env

```
MONGODB_URL = mongodb+srv://MERN:Mern123@cluster0.54hzvs3.mongodb.net/?retryWrites=true&w=majority
```

MongoDB Connection Ctd.

- Update server.js (index) file



```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const connectDB = require('./Database/connect');

require('dotenv').config();
connectDB(process.env.MONGODB_URL);

const app = express();
const PORT = 5000;
```

Validations in Database Operations

When you define a schema in Mongoose, you can specify various validation constraints on the fields, and Mongoose will ensure that these constraints are met before the data is saved to the database.

Type validation:
By specifying a field type (e.g., String, Number), Mongoose will ensure that the data being saved matches the expected type.

Required validation:
required: true
This will ensure that the field is provided before saving the document.

Validations in Database Operations

Unique validation:
unique: true
This ensures that the value is unique across all documents in the collection
Number-specific validators:
min and max: Set the minimum and maximum allowed values for number fields.
String-specific validators:

- minlength and maxlength: These set the minimum and maximum lengths for string values, respectively.
- enum: Allows you to specify an array of valid string values for the field.

Validations in Database Operations

Match pattern of a regular expression:

Eg: match: /^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/

- ^: Matches the start of a line.
- [\w-\.]+: Matches one or more word characters (\w), hyphens (-), or periods (.). This covers the local part of an email, which appears before the @ symbol. For example, in the email john.doe@example.com, john.doe is the local part.
- @: Matches the @ symbol.
- ([\w-]+\.)+: Matches one or more word characters or hyphens followed by a period (.). This is placed inside a group and followed by a + to allow for domain subdomains. For example, in the domain mail.example.com, mail. is a subdomain.

Validations in Database Operations

Match pattern of a regular expression:

Eg: match: /^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/

- [\w-]{2,4}: Matches between 2 to 4 word characters or hyphens. This covers top-level domains (TLDs) like .com, .net, .org, but may not include newer or longer TLDs like .design or .solutions.
- $: Matches the end of a line.

Custom validation:

Eg:

```
validate: {
  validator: function(value) {
    return value >= 18;
  },
  message: 'Age must be at least 18.'
}
```

Creating Database Schema Model

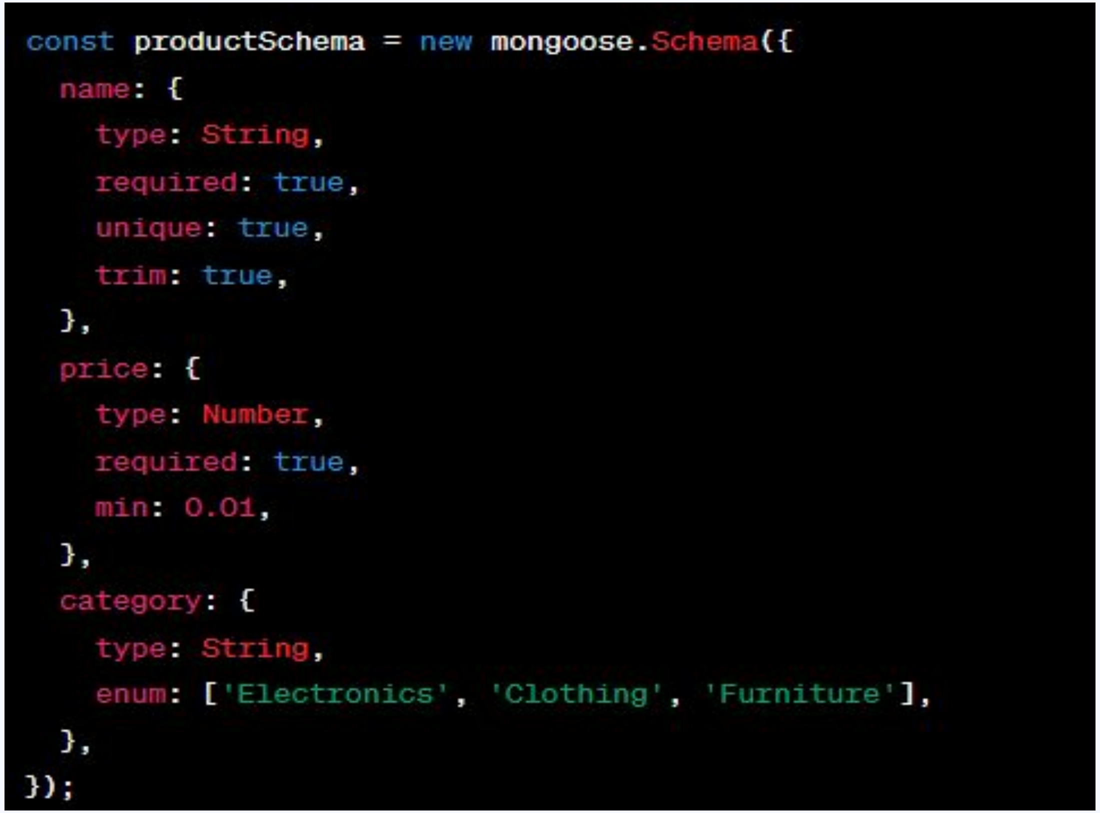
- Data in MongoDB has a flexible schema. Collections do not enforce document structure by default.
- Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections, by default, do not require their documents to have the same schema. That is:
    - The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- In practice, however, the documents in a collection share a similar structure, and you can enforce document validation rules for a collection during update and insert operations.

Schema Validation

- Schema validation ensures data consistency and integrity within the database.
- It helps prevent invalid or inconsistent data from being saved to the database.
- Validation rules are defined within the schema, reducing the need for manual checks in your application code.

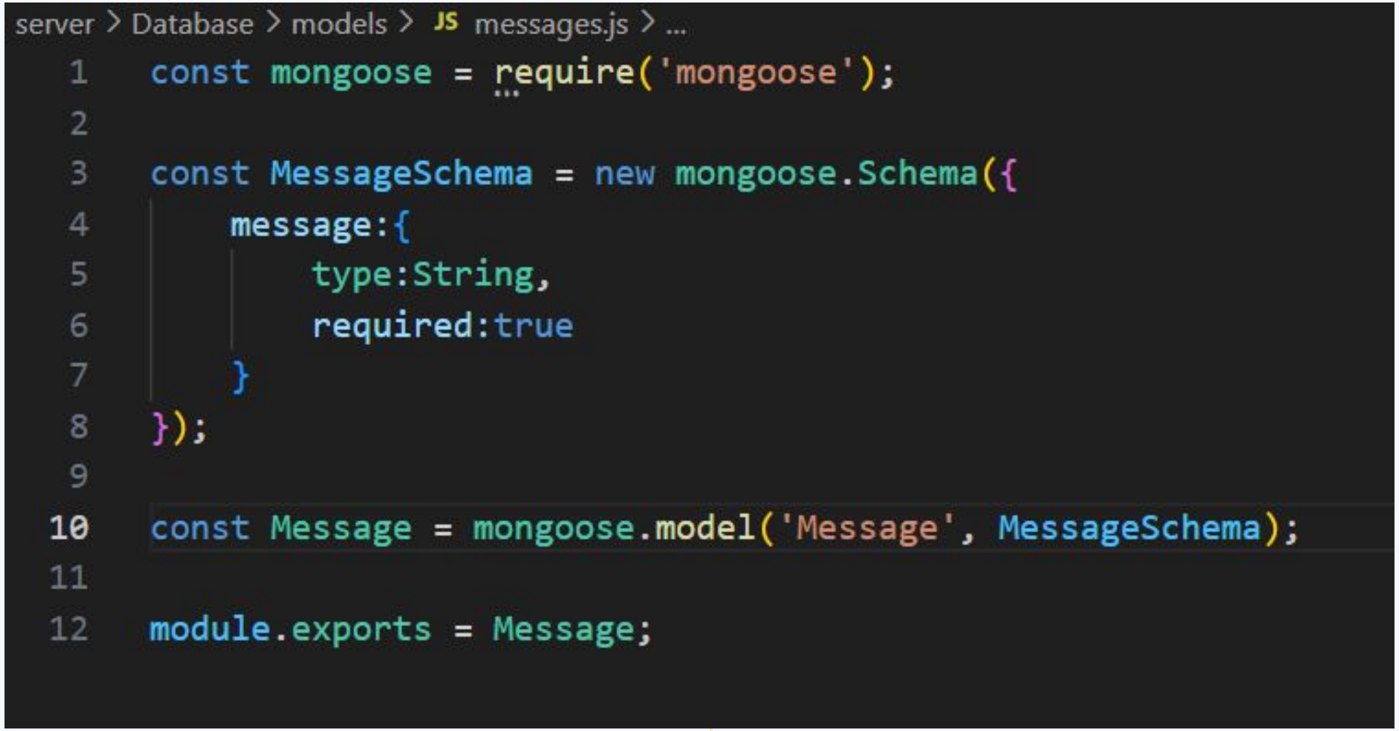
Key Features of Schema Validation

- **Data Types**: Define the expected data types for each field (e.g., String, Number, Date).
- **Required Fields**: Specify which fields must have values before saving a document.
- **Custom Validation Functions**: Implement custom validation logic for specific fields.
- **Enum Validation**: Restrict a field to a predefined set of values.
- **Min/Max Values**: Set minimum and maximum values for numeric fields.
- **Regular Expressions**: Use regular expressions to validate string formats.
- **Error Messages**: Customize error messages for validation failures.



```
const productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    unique: true,
    trim: true,
  },
  price: {
    type: Number,
    required: true,
    min: 0.01,
  },
  category: {
    type: String,
    enum: ['Electronics', 'Clothing', 'Furniture'],
  },
});
```

Creating Database Schema Model

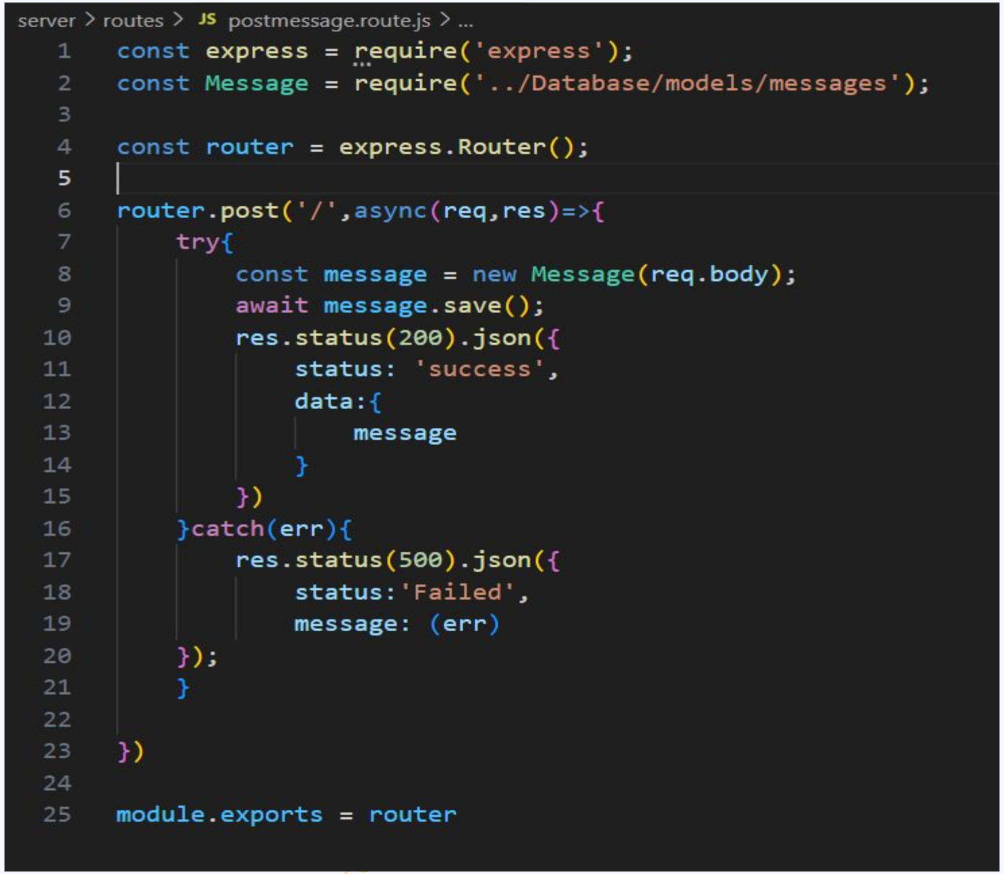


```
const mongoose = require('mongoose');

const MessageSchema = new mongoose.Schema({
    message:{
        type:String,
        required:true
    }
});

const Message = mongoose.model('Message', MessageSchema);

module.exports = Message;
```

Creating Database Schema Model ctd.

- The **mongoose.model()** function of the mongoose module is used to create a collection of a particular database of MongoDB.

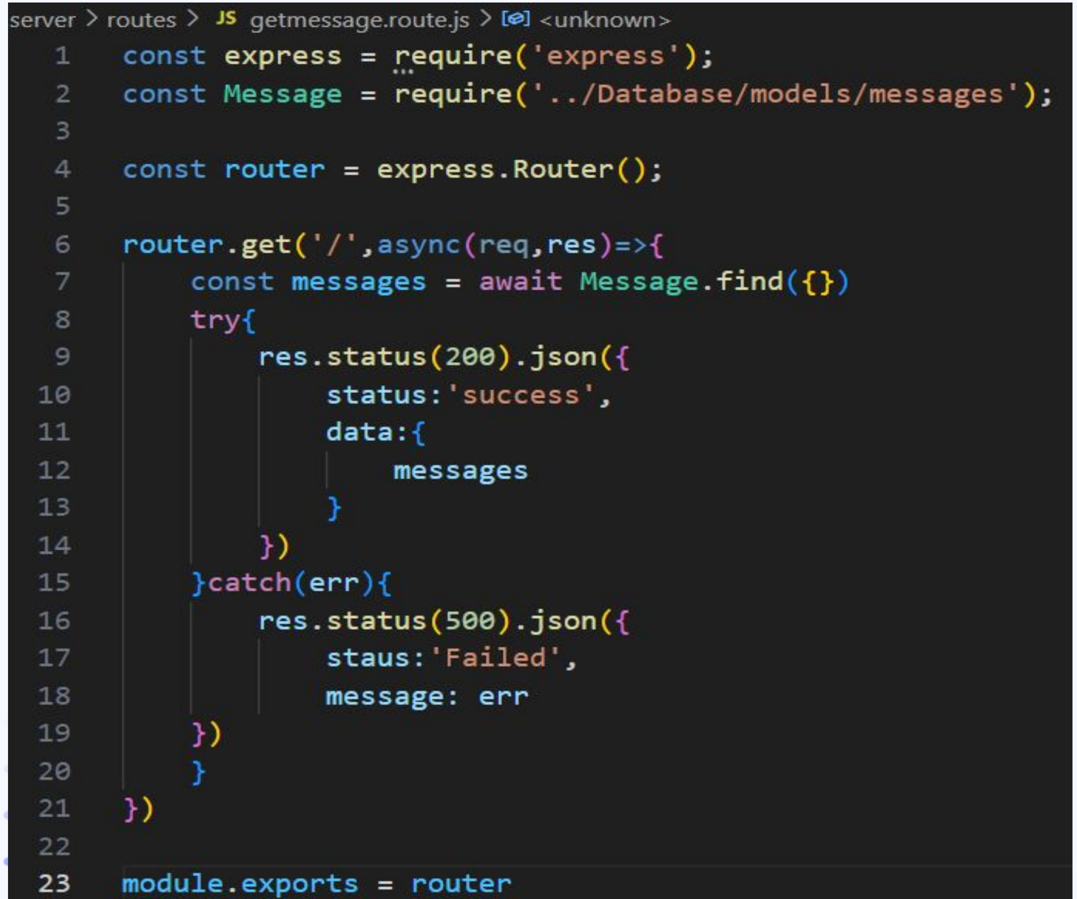
Creating APIs - Post Message



server > routes > JS postmessage.route.js > ...

```javascript
const express = require('express');
const Message = require('../Database/models/messages');

const router = express.Router();

router.post('/',async(req,res)=>{
    try{
        const message = new Message(req.body);
        await message.save();
        res.status(200).json({
            status: 'success',
            data:{
                message
            }
        })
    }catch(err){
        res.status(500).json({
            status:'Failed',
            message: (err)
    });
    }

})

module.exports = router
```

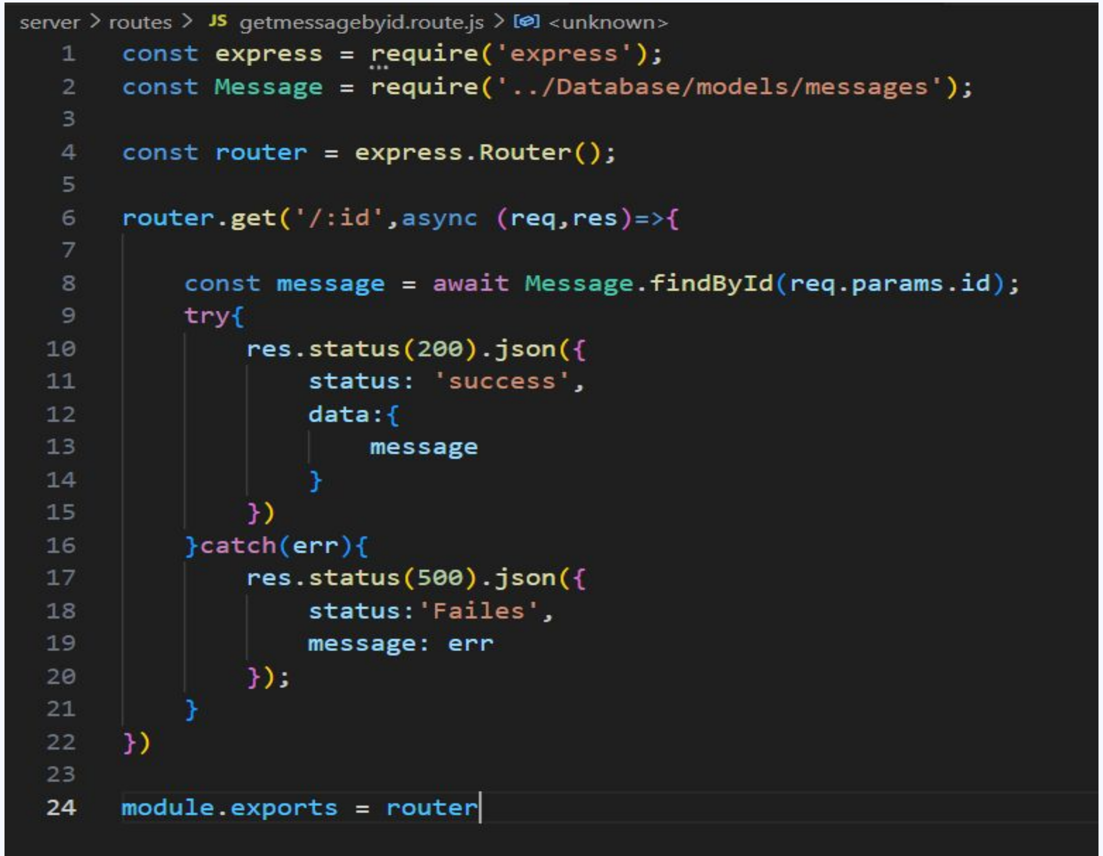
Creating APIs - Get Message



server > routes > JS getmessage.route.js > <unknown>

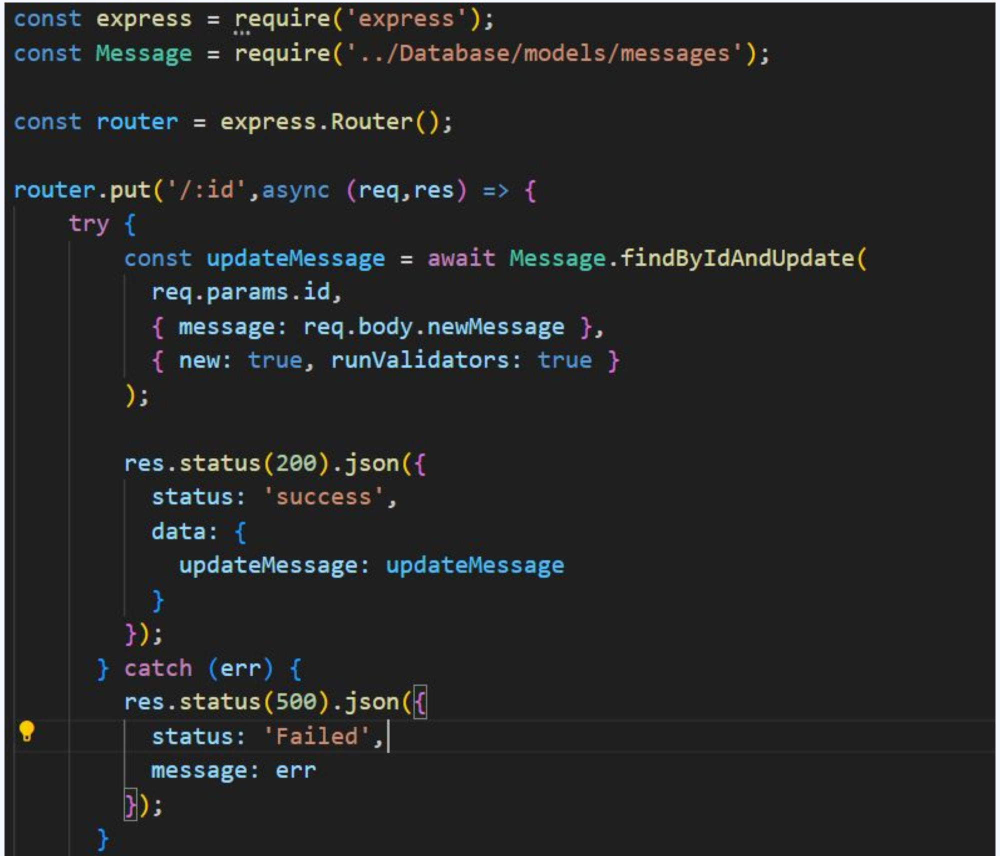
```
const express = require('express');
const Message = require('../Database/models/messages');

const router = express.Router();

router.get('/',async(req,res)=>{
    const messages = await Message.find({})
    try{
        res.status(200).json({
            status:'success',
            data:{
                messages
            }
        })
    }catch(err){
        res.status(500).json({
            staus:'Failed',
            message: err
        })
    }
})

module.exports = router
```

Creating APIs - Get Message by ID



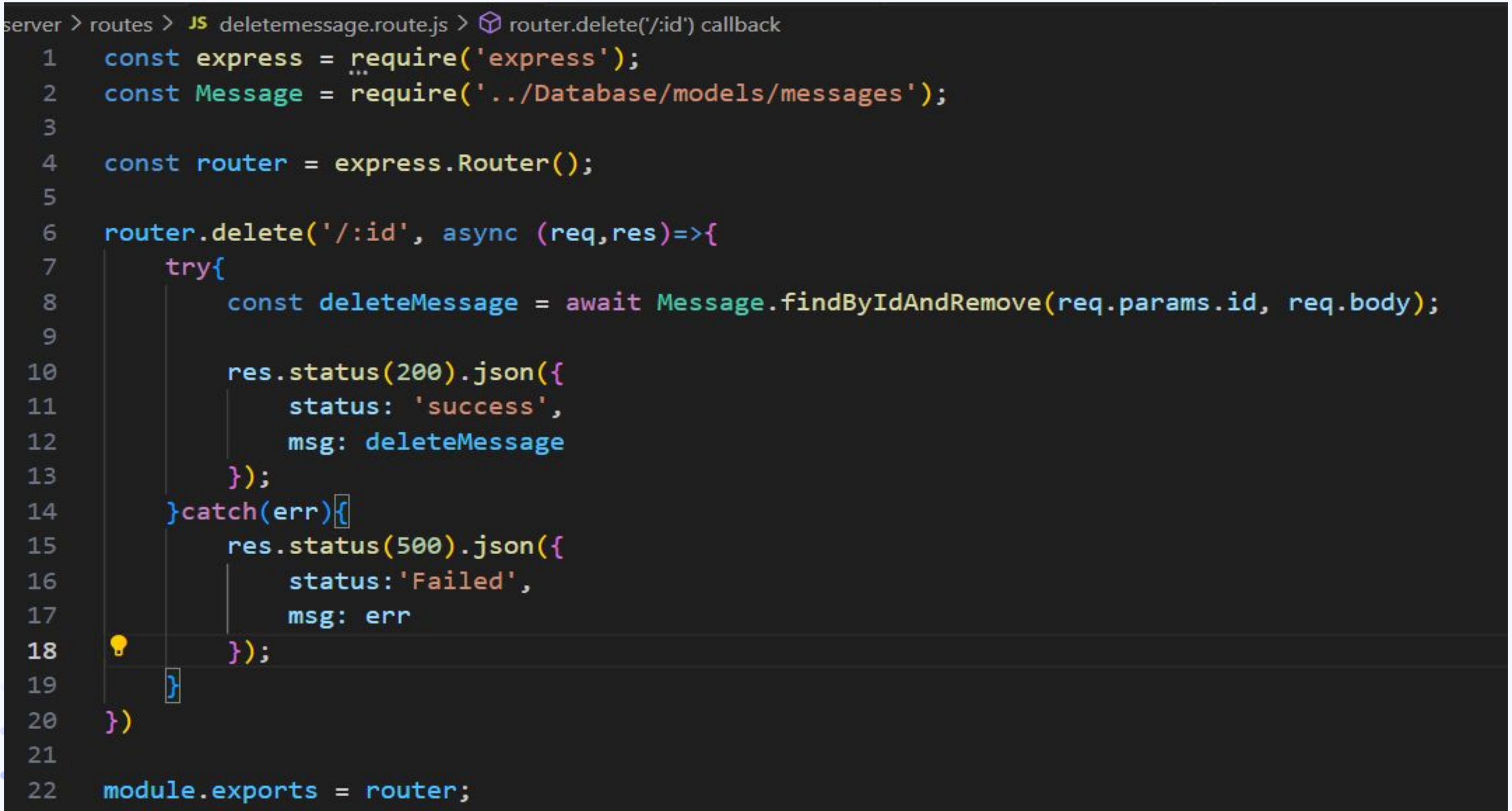
```javascript
const express = require('express');
const Message = require('../Database/models/messages');

const router = express.Router();

router.get('/:id',async (req,res)=>{

    const message = await Message.findById(req.params.id);
    try{
        res.status(200).json({
            status: 'success',
            data:{
                message
            }
        })
    }catch(err){
        res.status(500).json({
            status:'Failes',
            message: err
        });
    }
})

module.exports = router
```

Creating APIs - Update Message



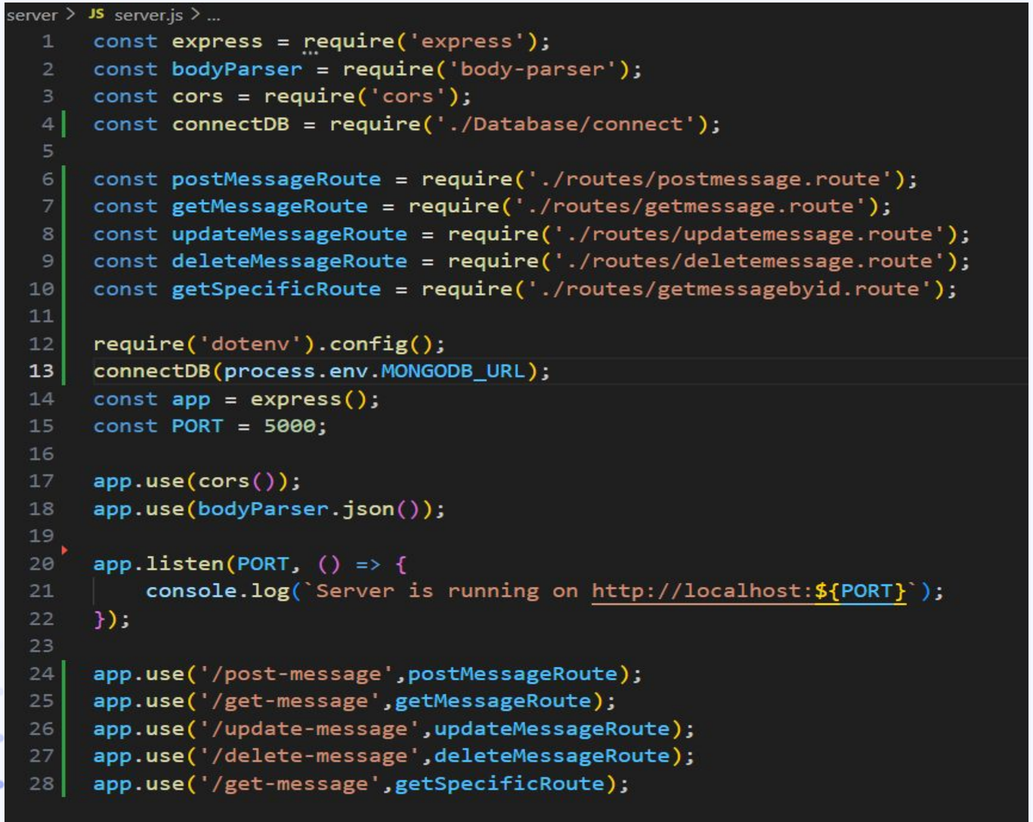
```
const express = require('express');
const Message = require('../Database/models/messages');

const router = express.Router();

router.put('/:id',async (req,res) => {
    try {
        const updateMessage = await Message.findByIdAndUpdate(
          req.params.id,
          { message: req.body.newMessage },
          { new: true, runValidators: true }
        );

        res.status(200).json({
          status: 'success',
          data: {
            updateMessage: updateMessage
          }
        });
      } catch (err) {
        res.status(500).json({
          status: 'Failed',
          message: err
        });
      }
```

Creating APIs - Delete Message


server > routes > JS deletemessage.route.js > router.delete('/:id') callback

```
const express = require('express');
const Message = require('../Database/models/messages');

const router = express.Router();

router.delete('/:id', async (req,res)=>{
    try{
        const deleteMessage = await Message.findByIdAndRemove(req.params.id, req.body);

        res.status(200).json({
            status: 'success',
            msg: deleteMessage
        });
    }catch(err){
        res.status(500).json({
            status:'Failed',
            msg: err
        });
    }
})

module.exports = router;
```

Using Routes/APIs



```javascript
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const connectDB = require('./Database/connect');

const postMessageRoute = require('./routes/postmessage.route');
const getMessageRoute = require('./routes/getmessage.route');
const updateMessageRoute = require('./routes/updatemessage.route');
const deleteMessageRoute = require('./routes/deletemessage.route');
const getSpecificRoute = require('./routes/getmessagebyid.route');

require('dotenv').config();
connectDB(process.env.MONGODB_URL);
const app = express();
const PORT = 5000;

app.use(cors());
app.use(bodyParser.json());

app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});

app.use('/post-message',postMessageRoute);
app.use('/get-message',getMessageRoute);
app.use('/update-message',updateMessageRoute);
app.use('/delete-message',deleteMessageRoute);
app.use('/get-message',getSpecificRoute);
```

Authentication

- Authentication verifies the identity of a user
- Example: Login page

Authorization

- Authorization determines their access rights
- Example: Only admins can delete posts, regular users can't.

Authentication

- Authentication is a process in which the credentials provided are compared to those on file in a database of authorized users' information on a local operating system or within an authentication server.
- If the credentials match, the process is completed and the user is granted authorization for access.

Authentication using Username & Password

1. Bcrypt: For hashing and verifying passwords.

   Why hashing?
   Hashing is a process of converting an input (like a password) into a fixed-length string of characters, which usually appears random.

   Purpose: It is used to protect sensitive data such as passwords from being accessed by attackers even if the database is compromised.

Characteristics of a Good Hash Function

- Deterministic: Same input will always produce the same output.
- Fast to Compute: Should be quick to hash data.
- Preimage Resistance: Given a hash, it should be computationally infeasible to find the original input.
- Small Changes, Big Impact: Even a small change in input should produce a significantly different hash.
- Collision Resistance: Two different inputs should not produce the same hash.

Bcrypt

- Salting: Bcrypt automatically handles the generation of random data and combines it with the password before hashing. This ensures that even if two users have the same password, their hashes will be different due to unique salts.
- Adaptive: The iterations of hashing can be adjusted, allowing the hashing process to remain slow and thus secure even as computers get faster.
- Ease of Use: Bcrypt is simple to use and automatically handles salt generation and management.

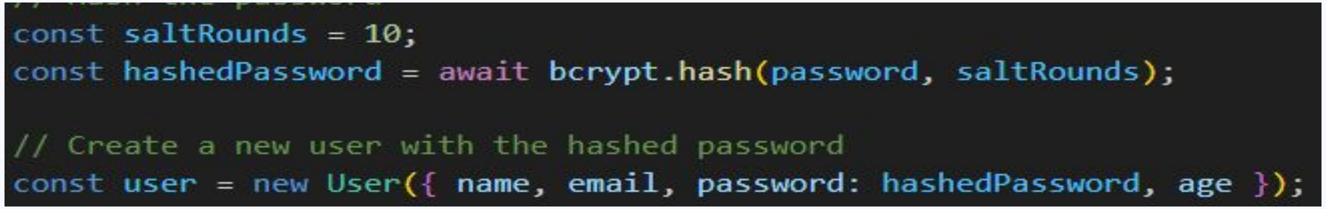
How Bcrypt Works

- Generate Salt: Bcrypt generates a random salt.
- Mix Salt and Password: The salt is combined with the user's password.
- Hash Multiple Times: The combined data is hashed multiple times (iterations) to produce the final hash.
- Store Result: The resulting hash, which includes information about the salt and work factor, is stored in the database.

Hashing password using Bcrypt Example

Install bcrypt package

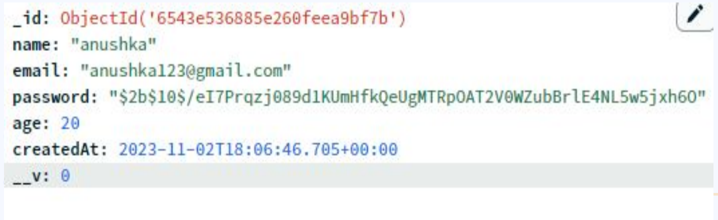
npm install bcrypt



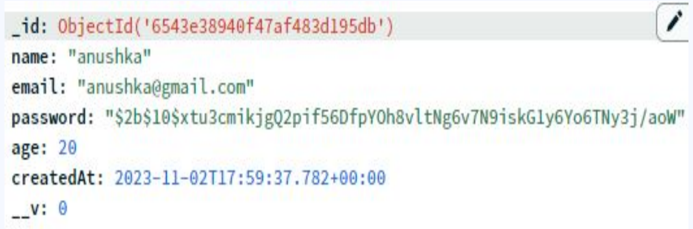
```javascript
const saltRounds = 10;
const hashedPassword = await bcrypt.hash(password, saltRounds);

// Create a new user with the hashed password
const user = new User({ name, email, password: hashedPassword, age });
```

A salt is a random value that is combined with the user's password before it is hashed. SaltRounds is set to 10, the bcrypt algorithm will perform 1024 iterations of hashing

Same password used but the hashed password differs



```
_id: ObjectId('6543e536885e260feea9bf7b')
name: "anushka"
email: "anushka123@gmail.com"
password: "$2b$10$/eI7Prqzj089d1KUmHfkQeUgMTRpOAT2V0WZubBrlE4NL5w5jxh6O"
age: 20
createdAt: 2023-11-02T18:06:46.705+00:00
__v: 0
```

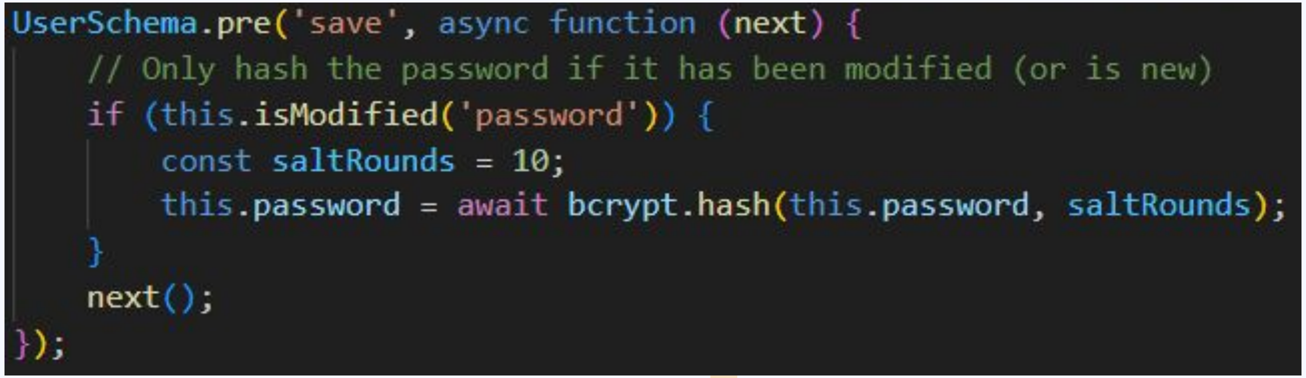


```
_id: ObjectId('6543e38940f47af483d195db')
name: "anushka"
email: "anushka@gmail.com"
password: "$2b$10$xtu3cmikjgQ2pif56DfpYOh8vltNg6v7N9iskGly6Yo6TNy3j/aoW"
age: 20
createdAt: 2023-11-02T17:59:37.782+00:00
__v: 0
```

Hashing password using Bcrypt Example

Defining in the schema

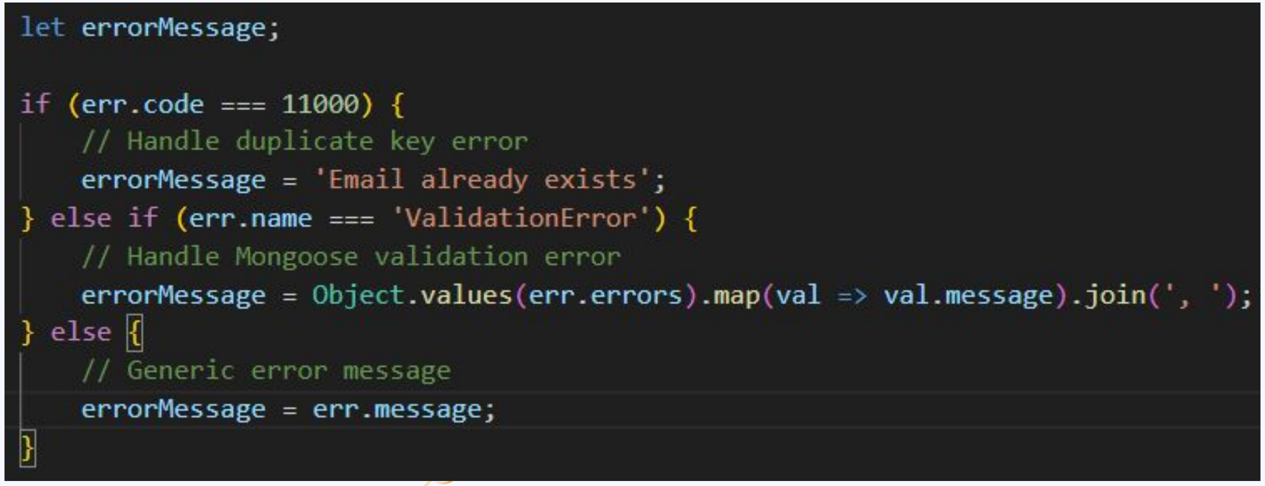
Before saving in database password is being hashed.



```
UserSchema.pre('save', async function (next) {
    // Only hash the password if it has been modified (or is new)
    if (this.isModified('password')) {
        const saltRounds = 10;
        this.password = await bcrypt.hash(this.password, saltRounds);
    }
    next();
});
```

Error Handling in Database Operations

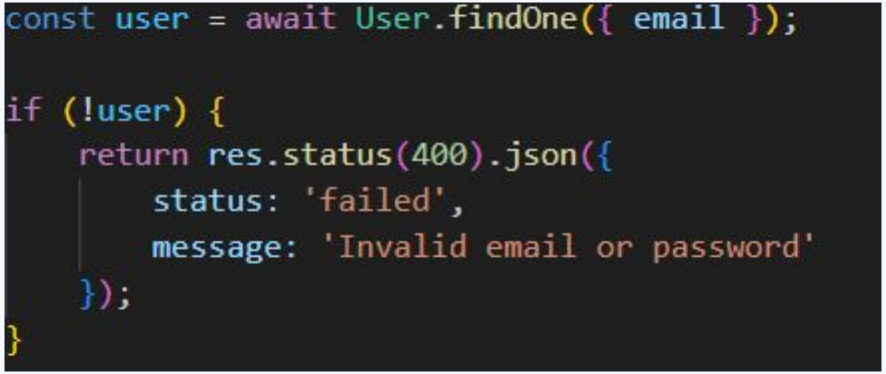
Mongoose provides a way to define validation rules on your schemas (e.g., a field must be a valid email, or a password must have a certain length). If any of these validation rules are not met when trying to save a document, a ValidationError will be thrown



```
let errorMessage;

if (err.code === 11000) {
    // Handle duplicate key error
    errorMessage = 'Email already exists';
} else if (err.name === 'ValidationError') {
    // Handle Mongoose validation error
    errorMessage = Object.values(err.errors).map(val => val.message).join(', ');
} else {
    // Generic error message
    errorMessage = err.message;
}
```

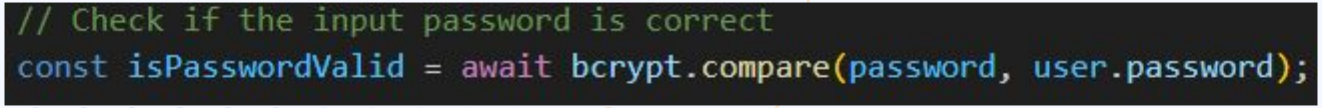
Verification

First check if the user exists or not with the given email



```
const user = await User.findOne({ email });

if (!user) {
    return res.status(400).json({
        status: 'failed',
        message: 'Invalid email or password'
    });
}
```

To verify the password, you can use bcrypt.compare() function to compare the plaintext password with the hashed password.



```
// Check if the input password is correct
const isPasswordValid = await bcrypt.compare(password, user.password);
```

JSON Web Tokens(JWT) authentication

- JWT is a compact, URL-safe way of representing claims to be transferred between two parties. It is often used for authentication and authorization in web applications.

- A JWT token typically consists of a header, payload, and signature. The header and payload are Base64Url encoded JSON strings, and the signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

How JWT authentication works

- User Login: When a user logs in using their credentials, the server validates the credentials.
- Generating JWT: If the credentials are valid, the server creates a JWT. This token typically encodes the user's ID and some other information. The server then signs this token using a secret key.
- Sending the JWT: The server sends the signed JWT back to the client. The client then stores this token. It could be stored in an HTTP cookie, or in local storage, or even in-memory, depending on the needs and security considerations of the application.
- Authenticated Requests: For subsequent requests that require authentication, the client attaches the JWT in the HTTP header (commonly the Authorization header with a prefix of Bearer).

How JWT authentication works

- Server-side Verification: Upon receiving the request, the server validates the JWT signature using the same secret key it used to sign it. If it's valid, it means the client is authenticated, and the server processes the request.
- Expiration: JWTs can have expiration times. If a token is expired, the server will reject the request, prompting the client to re-authenticate.
- Logging Out: Logging out can be handled by the client discarding the JWT it has stored. Additionally, the server can maintain a blacklist of tokens to ensure they can't be used again until they naturally expire.

Benefits of JWT

Statelessness: The server doesn't need to store session data. Each request contains all the information needed for processing.

Scalability: Since there's no need to store session data, applications can scale more easily.

Decoupling: The authentication server that generates the token can be separate from the server that processes requests, leading to decoupled, more maintainable architectures.

Use of JWT tokens

Install jsonwebtoken to work with JWT
npm install jsonwebtoken (server side)

In your .env file, add a secret for your JWT
JWT_SECRET=myVerySecretKey

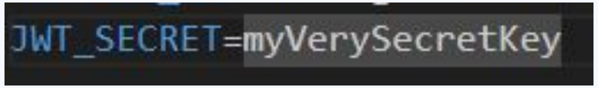
The secret key, sometimes known as the signing key, is used to sign and verify JWT tokens. When the server generates a JWT, it signs the token using this secret key.

Secret key

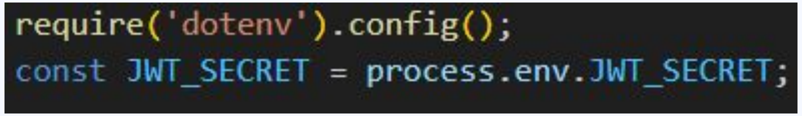
- Once the user logs in, the server creates a JWT token (signed with the secret key) that represents the user's session. This token is sent back to the client.
- The client (in this case, the frontend of your web app) then stores this token, often in local storage, so that it can be sent back to the server with every subsequent request. This way, the server knows and can verify who is making the request.
- During the verification process (when a request with a token is sent back to the server), the server will use the secret key to verify the token's signature. If the signature doesn't match (indicating the token might have been tampered with) or if the token is expired, the server can reject the request.

Generate the JWT token

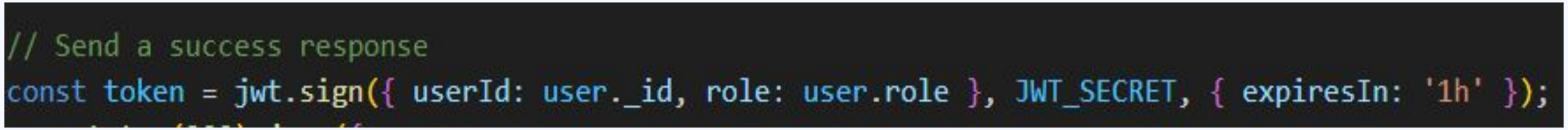
Modify your login route in login to generate a JWT token on successful user authentication.
It will be sent to the front end or client side as a response.



```
JWT_SECRET=myVerySecretKey
```



```
require('dotenv').config();
const JWT_SECRET = process.env.JWT_SECRET;
```



```
// Send a success response
const token = jwt.sign({ userId: user._id, role: user.role }, JWT_SECRET, { expiresIn: '1h' });
```

Generate the JWT token Contd

const token = jwt.sign({ userId: user._id, role: user.role }, JWT_SECRET, { expiresIn: '1h' });

Jwt.sign combines the input data and returns a concise and short URL, header with below parameters.

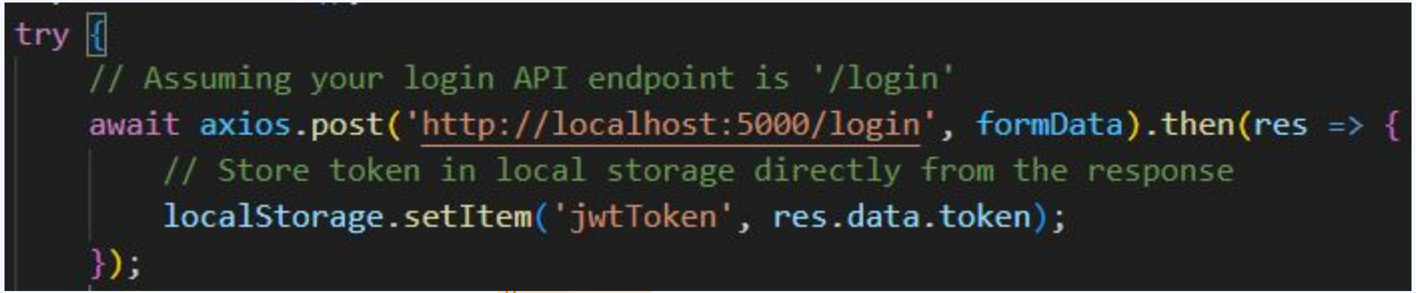
Parameters:

- **Payload ({ userId: user._id, role: user.role }):** The data you want to encode into the token.
- **Secret (JWT_SECRET):** A secret string used to sign the token and ensure its integrity.
- **Options ({ expiresIn: '1h' }):** Additional settings or configurations for the token.

Store the JWT token

In your Login React component, upon successful login, store the token in the local storage.

Local storage is a web storage solution that allows websites and web applications to store and access data as key-value pairs directly in a web browser. Unlike cookies or session storage, data stored in local storage is persistent. This means it remains available even after the browser is closed and reopened.

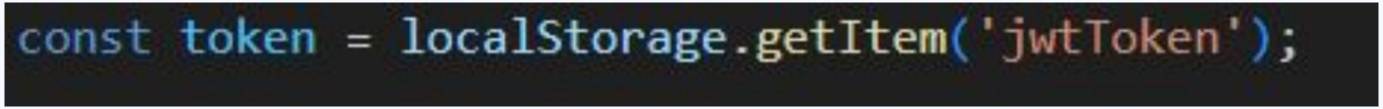


```
try {
    // Assuming your login API endpoint is '/login'
    await axios.post('http://localhost:5000/login', formData).then(res => {
        // Store token in local storage directly from the response
        localStorage.setItem('jwtToken', res.data.token);
    });
```

Role Based Login using JWT token

Different users (e.g., admin, user) should have different access rights and different permissions. So we will be using JWT token for this purpose.

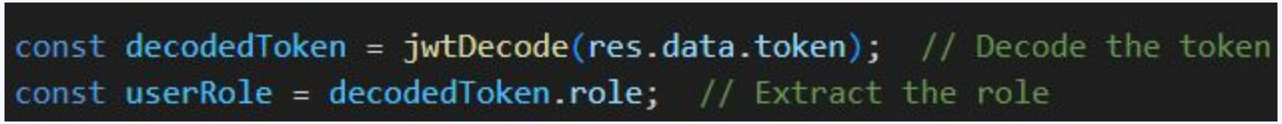
First we have to retrieve the token from frontend



```
const token = localStorage.getItem('jwtToken');
```

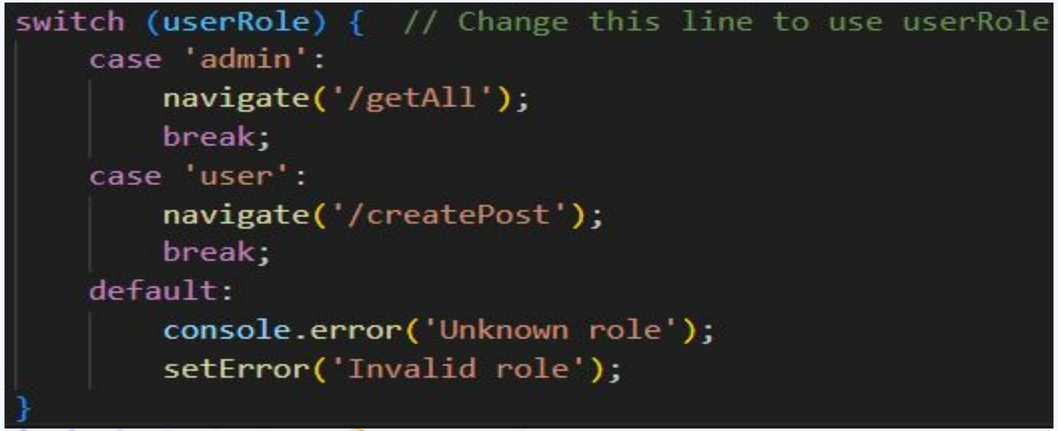
Next we have to decode or extract the data from retrieved jwt token. For that we'll be using jwt-decode library
npm install jwt-decode (client side)

Role Based Login using JWT token Example



```
const decodedToken = jwtDecode(res.data.token);  // Decode the token
const userRole = decodedToken.role;  // Extract the role
```

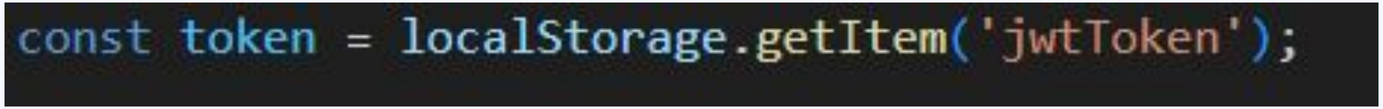
decodedToken object contains the data encoded in the token payload. For example, we have set the token in the backend with userID and userRole



```
switch (userRole) {  // Change this line to use userRole
    case 'admin':
        navigate('/getAll');
        break;
    case 'user':
        navigate('/createPost');
        break;
    default:
        console.error('Unknown role');
        setError('Invalid role');
}
```

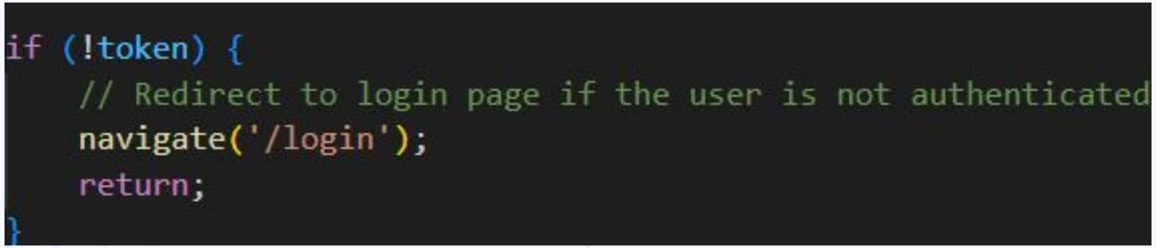
Sending a Request from Frontend with JWT token

First retrieve the token from localStorage



```
const token = localStorage.getItem('jwtToken');
```

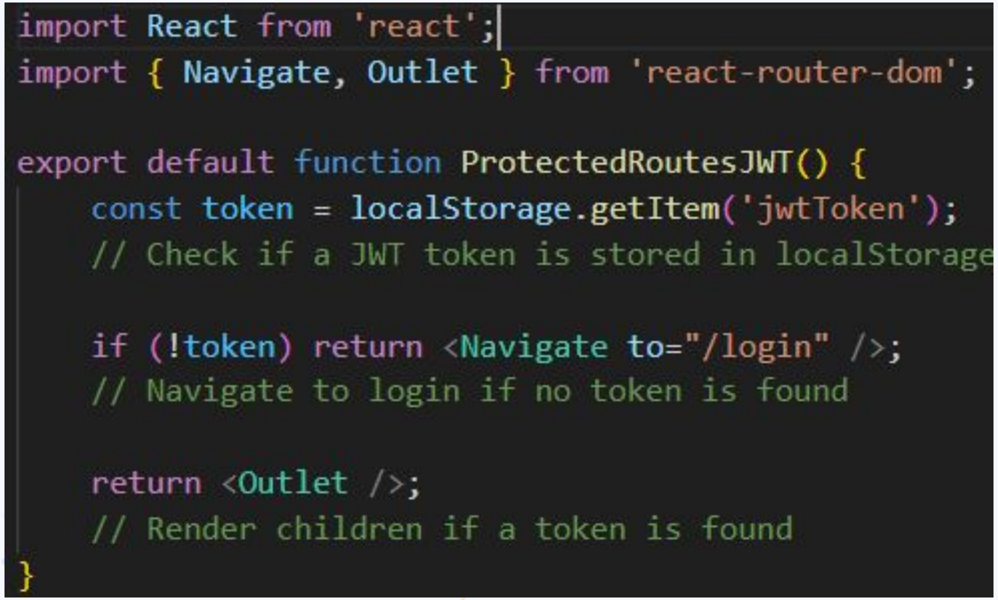
If the token is not set then redirect the user to the login page.



```
if (!token) {
    // Redirect to login page if the user is not authenticated
    navigate('/login');
    return;
}
```

Protected Routes with JWT

Here the concept is same as before, but we can use JWT token to check whether if user is logged into the system or not.



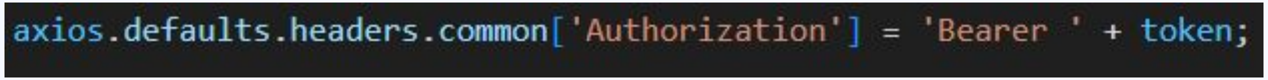
```
import React from 'react';
import { Navigate, Outlet } from 'react-router-dom';

export default function ProtectedRoutesJWT() {
    const token = localStorage.getItem('jwtToken');
    // Check if a JWT token is stored in localStorage

    if (!token) return <Navigate to="/login" />;
    // Navigate to login if no token is found

    return <Outlet />;
    // Render children if a token is found
}
```

Creating middleware auth for JWT Authentication in backend

For the Authentication and Authorization purpose using JWT we will be creating a middleware.

Middleware functions are often used to preprocess requests. For example, they can be used to parse the body of incoming requests, handle authentication and authorization, add headers, or even log data.
Middleware allows developers to control the flow of the request-response cycle. It provides a way to chain functions together, and depending on certain conditions, you can decide to end the response or continue with the next middleware.

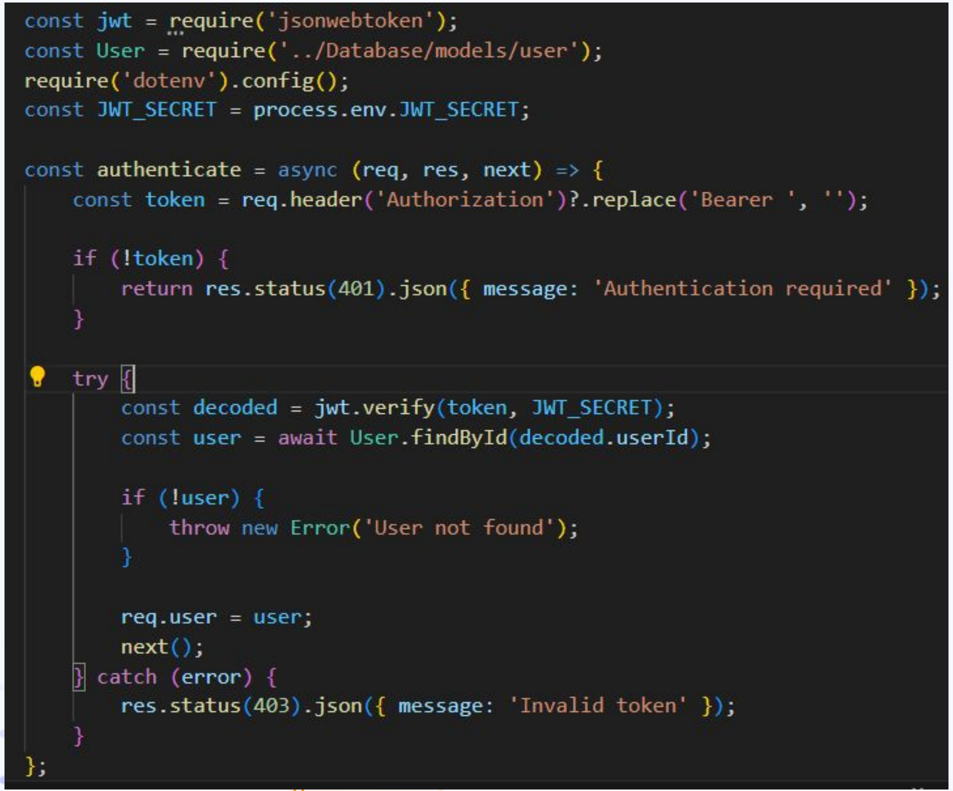
Sending a Request from Frontend with JWT token



```
axios.defaults.headers.common['Authorization'] = 'Bearer ' + token;
```

- This line sets the default Authentication header for all subsequent HTTP requests made using axios.
- Authentication Header: This is a part of the HTTP request sent from your app to the server. It includes a token which helps the server know who is making the request.
- Bearer Authentication: The token acts as a "bearer" of the credentials, meaning that whoever holds the token is assumed to have the correct permissions.

Creating middleware auth for JWT Authentication in backend



```
const jwt = require('jsonwebtoken');
const User = require('../Database/models/user');
require('dotenv').config();
const JWT_SECRET = process.env.JWT_SECRET;

const authenticate = async (req, res, next) => {
    const token = req.header('Authorization')?.replace('Bearer ', '');

    if (!token) {
        return res.status(401).json({ message: 'Authentication required' });
    }

    try {
        const decoded = jwt.verify(token, JWT_SECRET);
        const user = await User.findById(decoded.userId);

        if (!user) {
            throw new Error('User not found');
        }

        req.user = user;
        next();
    } catch (error) {
        res.status(403).json({ message: 'Invalid token' });
    }
};
```

Creating middleware auth for JWT Authentication in backend Contd

const token = req.header('Authorization')?.replace('Bearer ', '');

In Express.js, the **req.header() function** is used to get the value of a specific HTTP header from the incoming request.
? - Conditional operator in JS, if header exists then do what's after it
.replace('Bearer ', ' '): The **replace** method is a string method that searches for a specified value, or a regular expression, Here, it's looking for the string **'Bearer '** and replacing it with an empty string.

Eg:
Receiving - **Authorization: Bearer eyJhb...**
After Extraction - **eyJhb...**

Creating middleware auth for JWT Authentication in backend Contd

const decoded = jwt.verify(token, JWT_SECRET);

The verify method does two main things:

- Checks if the token's signature is valid using JWT_SECRET, The signature is a part of the token that proves it hasn't been tampered with
- It decodes the payload of the JWT token.

const user = await User.findById(decoded.userId);
Decoded variable contains the payload data of the JWT token, and querying the database for a user with an ID that matches **decoded.userId**

Creating middleware auth for JWT Authentication in backend Contd

req.user = user;

New property user is being added to the req (request) object and it's being set to the user that was fetched from the database in the previous line

This is a common practice in middleware functions that handle authentication. By attaching the user object to the req object, subsequent middleware functions and the final route handler can access the authenticated user's data without needing to fetch the user again from the database.

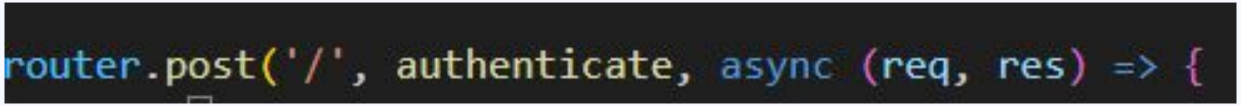
Creating middleware auth for JWT Authentication in backend Contd

next()

The next function is a part of the Express.js middleware architecture. When called, it passes control to the next middleware function in line, or if there are no more middleware functions, to the route handler.

In the context of this authentication middleware, calling next() means "this request has been successfully authenticated, and we can proceed to the next step in processing this request."

Authenticating the routes

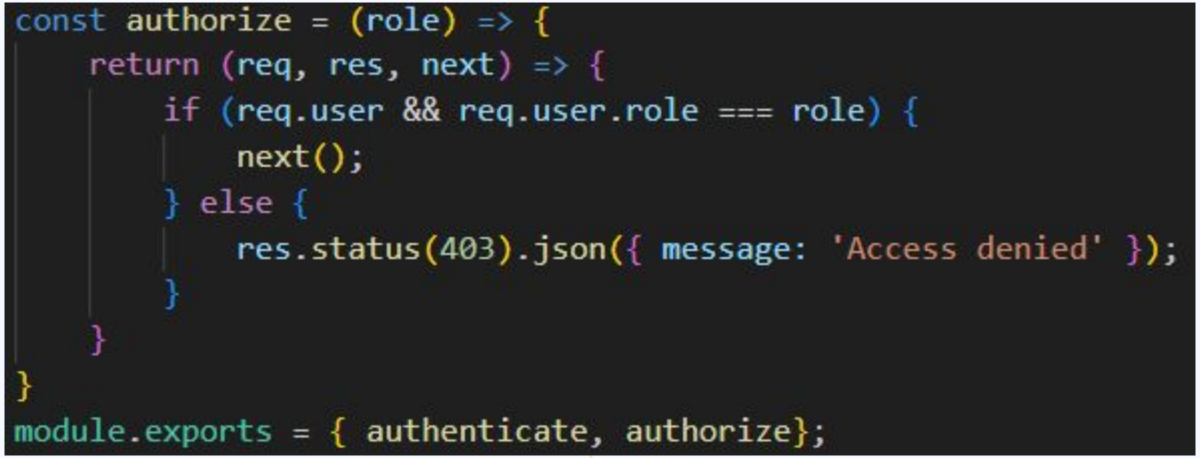

```
router.post('/', authenticate, async (req, res) => {
```

Authenticate is the middleware function we discussed earlier. By placing it as the second argument, before the actual route handler, you're saying "before processing the POST request with the route handler, first run the authenticate middleware".

- If the authenticate middleware successfully completes (i.e., the user is authenticated and the next() function is called inside it) the rest of the function executes.
- If there's an error in the authenticate middleware (e.g., the JWT token is invalid), the route handler will not be executed.

Creating middleware auth for JWT Authorization in backend

In the same middleware we created for Authentication we'll include the Authorization as well.



```
const authorize = (role) => {
    return (req, res, next) => {
        if (req.user && req.user.role === role) {
            next();
        } else {
            res.status(403).json({ message: 'Access denied' });
        }
    }
}
module.exports = { authenticate, authorize};
```

Creating middleware auth for JWT Authorization in backend Example

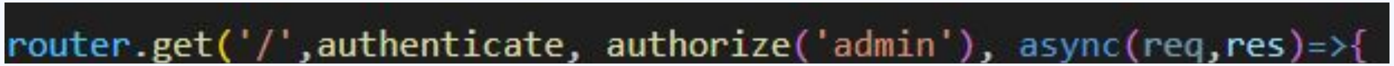
const authorize = (role) => {

There is a parameter role which is being given from an outer function when calling authorize function

Then the inner function (req, res, next) is the actual middleware that will be executed on a request.

if (req.user && req.user.role === role) checks:
Checks if there's a user object in the req (request) which is added before in authentication middleware and checks if the role of the user matches the required role passed to the authorize function.

Authorizing and Authenticating the routes



```
router.get('/',authenticate, authorize('admin'), async(req,res)=>{
```

Authenticate would verify if the user is logged in and attaches the user object to the request (req.user).
Authorize('admin') checks if the authenticated user has the role of 'admin'. If not, access is denied.

Thanks!